Visio Rules Engine Design Doc

Problem

Create an extensible rules engine that, when given a person and product, can iteratively apply rules that ultimately generate a final interest rate for that customer/product combination.

Excerpt:

"You have been tasked by a stakeholder at Visio Financial Services to develop a solution that allows the business to dynamically generating product pricing from a set rules defined by the finance team. The finance team has given you an initial set of rules on how to price the products, however, these rules could change at any time so we need to be able to update the rules easily and rerun the product pricing to see the new prices of the products.

Initial Rules: All products start at 5.0 interest_rate. If the person lives in Florida (condition), we do not offer the product to them and the product is to be disqualified (action). If the person has a credit score greater than or equal to 720(condition) then we reduce the interest_rate on the product by .3 (action that has an input of ".3", remember the business may decide in the future they want to reduce it by .5). If the person has a credit score lower than 720 we increase the interest_rate on the product by .5. If the name of the product is "7-1 ARM" then we need to add .5 to the interest_rate of the product."

Considerations

1. **Frequency of rules updates.**

As a given constraint, the rules engine needs to be easily updatable. This leaves some room for interpretation on how urgent the rules updates are. The importance of these updates impacts the design of the engine. There are a couple ways we can treat the updates:

Scenario A – Updates are incredibly urgent and should update the engine immediately. In this scenario, the application could automatically re-query the rules repo every time it begins calculating a new person-product interest rate to ensure the rules are always fresh. Since we would be querying the rules repo for every client request, we can trim down the rules request to only return rules that apply to the product at hand, instead of loading the entire rules set. The smaller rule set and smaller query would decrease the time and memory required to return the computed interest rate. That being said, we need to know more about the anticipated rule-set collection. Is this intended to be thousands of rules, or can we assume that there might only be dozens of rules needed. In the latter case, we wouldn't be saving a tremendous amount of memory by only loading a subset of the rules (since the total rule-set is small relative to the available resources on a typical server) and we could skip the optimization of loading partial rules sets. One of the downsides here is that we have to reload the rules every time we generate an interest rate, so we would want to make sure this cost is worth the freshness-factor that we gain.

Scenario B – Updates are important but not urgent. In this situation, it may be appropriate to refresh the rules on a specific time interval. Since we are currently only servicing loans in the US market (**assumption**), we can get away with reloading the rules outside of business hours. A 24-hour lag on rules being applied may be acceptable. In this scenario, we could have a scheduled script that safely reloads the rule-set. By safely here, I mean that we need to take extra care to make sure the engine is not actively in use. In this situation, we could potential have clients encounter down-time when the server is not responsive to requests. Regarding active rules sets, we could maintain a large rules set in memory and require the user/application explicitly invoke a 'refresh rules' method. The benefit of this is that we could share a single rules engine to process a queue of client requests. This could scale very nicely to thousands of queries per second on a single machine (though some redundancy is warranted to avoid single-points of failure). If the client base were to grow substantially, we could design a system to have multiple engines using the same rules set and processing from the same queue. Architecturally, we could have the queue sitting on a separate server between clients and the rules engine that can collect requests while we restart (or pause and reload rules for) the rules engine. The queue is an interesting architectural option for us to evaluate, but I will leave it outside of the scope of this problem for now by **assuming** we will only be running the rules engine during US business hours and the current scale does not require multiple rules engines to meet demand, meaning a single application can listen for and fully process the client requests. One neat aspect that we can put into designing for this scenario is that by having programmatic access to reloading rules, we allow ourselves the opportunity to manually load rules at a non-scheduled time if an urgent update is required. One example could be rolling back a mistake. Let's say a new rule was applied that was generating incorrect rates for a substantial number of clients. We could use the same code that restarts the engine with an updated rules set, but use it to load the previous rules set from the day prior. This brings up a consideration of rollbacks.

For our scope, we will **assume** scenario B and design the rules engine to use a single rules set until a new rules set is explicitly loaded (either by a scheduled script, or manually updating the rules set).

2. Rollbacks in Production

Another important aspect of this design has to do with the rolling back to a previous rules set. The implementation of archiving previous data sets is outside of the scope of this problem, but should definitely be considered for the end-to-end solution. In production systems that I have rolled out in the past, there were two approaches:

Approach A –Maintain the live application, and keep a copy of the most recent version for rollbacks. This release cadence was less common and the release itself underwent substantial automated and manual testing before release.

Approach B – Maintain the live application and a previous stable version. A version got moved to stable after being live for several months of maturation to prove the stability of the application and its new feature set. The latest version is where we published our daily builds for clients to validate and provide feedback on. We would pay close attention to make sure clients were using the version that was appropriate for their use case.

Regardless of approach, rollbacks will be supported by simply reloading the previous rule set into the engine.

3. Security

Much of the evaluation and execution of rules is dynamic in nature to allow easy extensibility. Great care should be taken to protect the production rule set as this is an opportunity for malicious instructions to be fed into the rules engine. The repo for production rules should have as restricted write access as possible and have a stringent test and validation process before anything is added to it.

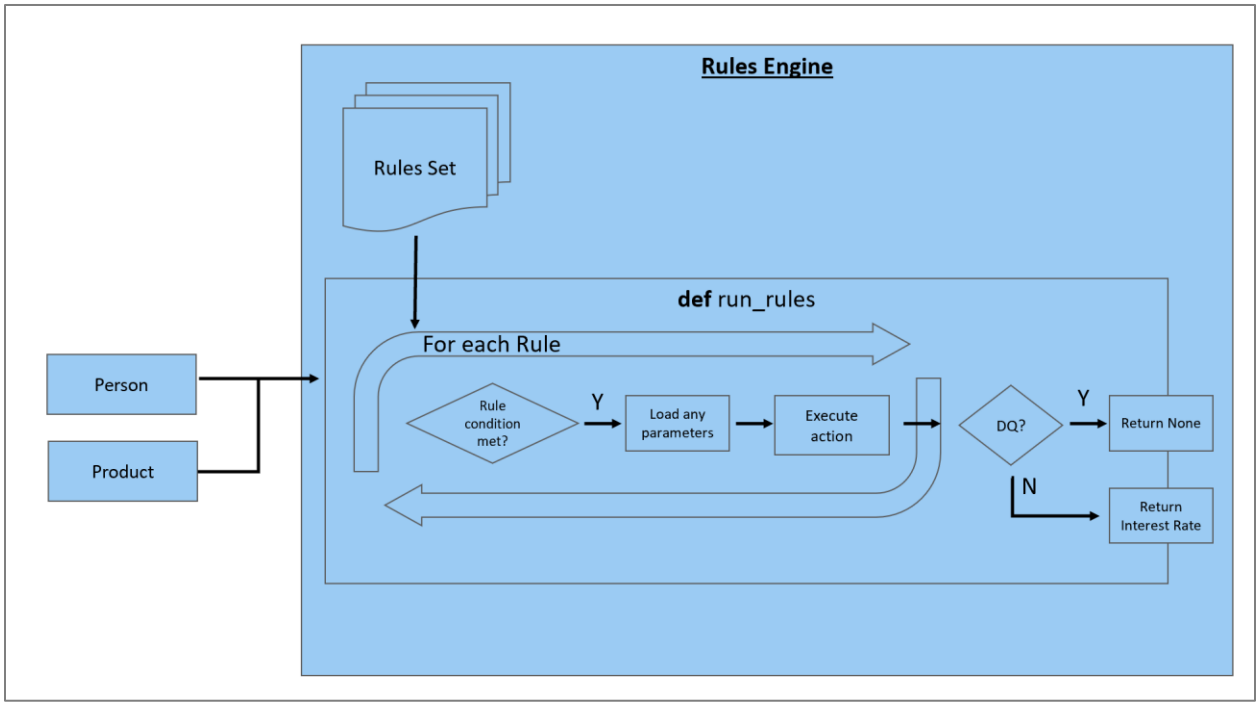
Design

The product and persons class should be pretty straight-forward and are not the focus of the exercise. Below we will discuss the rules engine and the rules themselves.

1. Rules Engine

The rules engine will be a relatively simple class with one attribute – the active rules set – and two methods for running and reloading rules. I have chosen to include the rules set in the rules engine instance intentionally to allow a single engine to run many client queries while using the active rules set. This is an assumption further described in number 3 of the Other Assumptions section.

The run_rules method is the heart of the engine. The structure of the method is to iterate through each of the rules, evaluate their condition, and if the condition is met, load any parameters and execute the action. This will continue until there are no rules left, then either return the product interest rate or None if the product has been disqualified.



In the initial rules set as well as the additional rule I added, the order of execution of the rules did not matter. This was because no actions were dependent on previous results. It is possible that order may matter in the future – e.g. "if the final interest rate is >10%, disqualify the product and don't offer it to the customer". This could be immediately accommodated by deliberately maintaining the order of the rules array when rules are added, but we could later extend the system to include a priority as part of each rule. This priority could be an integer or an enum and will allow us to more easily add rules as the rules set scales (i.e. we can just add the rule with its priority instead of trying to figure out where it belongs in a rules array with 100 elements).

2. Rules

The defined components of a rule are the action (referred to interchangeably in this document as the callback), the parameters for the action, and the conditional for the action. We will discuss each individually here:

Action – The examples list two types of actions: an addition or subtraction to the final interest rate, or a disqualification for the product. We will need to handle both of these types of actions. It is worth acknowledging that rules could behave in fashions outside of these two types. One example could be using multipliers, which should be easily extensible with our callback approach. We will try to accommodate for other unforeseen behaviors by modularizing the callback to be any executable piece of code with explicit access to the context of the current person and product being evaluated.

The action itself will be in the format of one or more python statements stored as a string. We will use the exec() function in python to feed these action statements dynamically into the python interpreter.

This should allow significant flexibility into what actions the rules can execute, but does bring certain security risks if the actions being fed into production are not strongly protected. It also puts requires rules writers to be familiar with python. One way around this would be to wrap rules generation in a user-friendly fashion (see Rules Wizard below).

Conditions – The example conditions are all referring to attributes of the person or product in question. With this in mind, we will store the expressions as a string following python expression syntax and use the eval() function to evaluate them dynamically. These conditions will have access to relevant contextual data of the person and product to use as necessary. This requires extreme care and diligence when writing the rules to make sure they are syntactically correct, match the definitions of the Person and Products, and can be dynamically evaluated by the python interpreter. One option to address this concern is again with the rules wizard proposed below. Alternatively and in addition to the wizard, all rules should be tested before being pushed to production, which should catch both behavioral and syntactical errors.

Parameters – parameters will be implemented as a dictionary to provide more comprehensible code compared to *args (e.g. "*rate = params[base_rate]*" is more understandable than "*rate = args[0]*"). These will be stored and read in as raw json strings and converted into a python dictionary.

It is unclear in the description what the scope is of the parameters. If it is limited to the person and product in question, our engine already has that information available when executing rules. It is also possible that all of the parameters are static values, as given in many examples. These values are known at development/configuration time. Static values are handled by the current implementation and person/product attributes are directly accessible by the actions without being read in from the parameter dictionary.

One other interesting possibility in the future would be allowing even more dynamic parameters that are determined completely at run-time. For example, we may want to alter an action depending on the current state of the market (certain stock indicators or rates set by the federal reserve). This kind of information could be readily and programmatically accessible with internet access. With minor adjustments, the current implementation should have the technical ability to extend to this by allowing the parameters themselves to execute python code using the same mechanism present in the Action execution.

Overall, the final format of the rule is an array of Rules similar to the example below:

```
{ # Good Credit Incentive
            'id' : 2,
            'condition' : 'person.credit_score >= 720',
            'action' : 'product.interest_rate += params["interest_rate_delta"];',
            'parameters' : '{"interest_rate_delta":-0.3}'
        }
```

3. **Rules Wizard (optional)**

Outside of scope, but it would be nice to create a wizard that streamlines the creation of new rules. I anticipate the formatting and spelling of the rules being very fragile to manually create. The parameters, conditionals, and callbacks will need to match existing definitions of classes and be syntactically correct.

This could be implemented a number of different ways, but a simple HTML page with a section for each of the parts of the rule would be quite helpful. Taking it one step further, it could include a section to allow the creator to interactively test the outcome of the rule individually or in combination with all other existing rules. Finally, this wizard could potentially add the new rule to a queue to be deployed to production. If this last bit of functionality is desired, we would likely want to include some sort of approval either before the rule gets added to the queue or before the queue gets pushed to production.

Other Assumptions

Many assumptions are marked throughout this document, but some did not belong in particular sections or warranted further elaboration:

1. I assumed the first rule “all products have an initial interest of 5.0” can be handled in the product constructor instead of as a rule. To change this to be a rule, it could be added to the rule set with a product rate impact of +5, while also changing the constructor to initialize the base rate at 0 instead of 5.

2. Paraphrased from the initial rules set, “if the person lives in florida, we do not offer them the product and the product is to be disqualified”. I am assuming that this means the product is only disqualified for that person as opposed to the product being disqualified universally to all applicants. This assumption is based on the idea that we will reuse the Rules Engine as well as the Product instances for many queries. In pseudo code:

```
15yr = Product(name=”15 year mortgage”)

tom = Person(credit_score=740, location=”Florida”)

betty = Person(credit_score=740, location=”Nevada”)

rules = load_rules()

engine = RulesEngine(rules)

engine.run_rules(tom, 15yr)          # returns None since disqualified

engine.run_rules(betty, 15yr)        # returns 4.7 (5.0 base rate - 0.3 for good credit)
```

3. This is a bigger assumption as it goes against the example in the prompt, but I believe that the rules set should be universal for a particular instance of a Rules Engine. These rules can be updated, but requiring rules as a parameter to the run_rules function seems unnecessary as we are likely applying a single rules set to all queries at a single point in time. Phrased in common terms, Jeff should get the same rules set as Erica who should get the same rules set as anyone

else who applies. There are perhaps some instances where potential customers live in a different rules set entirely – one that can't be accomplished by a single rule that could otherwise be added. Potential unique rules sets might be something like Veterans/Active Military, or Teachers. As stated prior, for simplicity I am going to assume that everyone will use the same rules set at any point in time. If this assumption is incorrect, I would add a parameter to run_rules for the application to pass in a rules set and everything else should run smoothly.

Example Usage

Below is an example of using the rules engine:

```
import visio


Luke = visio.Person(850, 'Texas')
product = visio.Product('15 year')
rules = visio.load_rules()
engine = visio.RulesEngine(rules)

interest_rate = engine.run_rules(Luke,product)
if interest_rate is not None:
    print(f'Congrats, your approved interest rate is {interest_rate}!')
else:
    print(f"Unfortunately, we can't approve your request at this time.")
```

Testing

Both unit and system tests were run on the rules set. The unit tests confirmed that each rule worked in a vacuum as if it were the only rule in the set. System tests went through every permutation of rule combinations with a variety of parameters. This covers the rules behavior performing as expected.

Some failure mechanisms that could exist is if we get unexpected values within the Person and Product instances. When integrating with the more holistic system, there should be validations put on the various parts of the Person and Product objects to improve the overall robustness of the system (credit scores should be within 300-850, location could be a state drop down, etc). These could be prioritized as necessary depending on how the overall system is setup and how the engine is consumed. These failure mechanisms and error cases are important, but without understanding the context of the system, it is unclear where to invest in additional testing and error handling.